Exercise 2: Generating Iterated Images

EECS 111, Winter 2017

Due Wednesday, Jan 25th by 11:59pm

Introduction

In this assignment, you will experiment with generating images using the `iterated` library. Throughout the assignment, you will:

- Practice writing functions with and without names (also known as *named* and *anonymous* functions),
- Practice translating mathematical expressions into code.

One of the core principles in computer science is the notion of **abstraction**, or masking the nitty-gritty details of how something works. This assignment will give you practice with the process of abstracting programs. You'll begin by writing very basic code, then gradually build up layers of abstraction to mask complexity, minimize repetition, and write reusable and generalizable patterns.

We've provided some starter code for you in the `Exercise-2.rkt` file, so open that up to get started.

Getting Started

Start by **running the file** once to make sure that the `require` commands successfully import the image libraries we're using.

When you run the file for the first time, a window will pop up saying that all of your tests failed. **Don't panic.** You haven't typed any code yet! Just close the test report window and carry on. (In the future, though, you should always read the window before you close it.)

To experiment with this assignment, you have two options. You can work directly in the **Definitions Window** (the main part of the DrRacket editor). Whenever you want to test what you've written, just run the file, which will generate a testing report in a popup window.

> *Tip: If you want to work on one part of the homework at a time, **select any tests you don't want to run**, and click `Racket > Comment Out With Semicolons`. Racket will skip over your comments when it runs your program, so you won't get notified if those tests fail. When you're ready to tackle those parts of the assignment, highlight the commented-out region and click `Racket > Uncomment` to restore the code. **Don't forget to uncomment and run all your tests.***

Alternatively, you can work in the REPL (aka the **Interactions Window**), the part of DrRacket with the `>` at the bottom of the screen. The REPL is nice for low-commitment fiddling, and you can hit `Ctrl-Up` to repeat your previous command if you want to make adjustments. Keep doing this until you get the result you're looking for. Once you're satisfied with your results, paste your code into the Definitions Window in the appropriate place.

Testing

The starter code contains a series of **unit tests**, which are pieces of code that test your code against the expected result. When you run your code by clicking the "Run" button or `Ctrl+R`, DrRacket will run the tests and generate a report of what tests succeeded and failed. If any tests failed, they'll pop up in a window. *Read the window* to see what happened, do *not* just close it! **Again: do not just close the window!**

To debug your output, you'll need to compare your image results to the expected results. **If you have a visual impairment and need an accommodation for this assignment, don't hesitate to email Sara.**

Getting Help

Remember that every function used in this assignment has documentation. If you're confused by a function, type it into DrRacket, right-click the name, and select "Search in help desk" to find an explanation. **Due to a bug, you need to first clear the search parameters by clicking the "Clear" button next to "Intermediate Student with Lambda." Otherwise, you won't see documentation for the functions in `cs111/iterated`.**

You can also post on Piazza or attend one of the peer mentor hours (we have a lot).

Part 1: Iterative Expressions

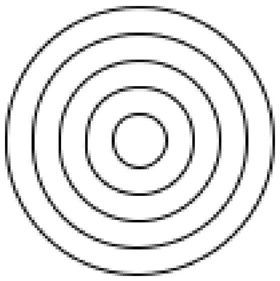
In this section, we'll use three iterator functions: `iterated-overlay`, `iterated-beside`, and `iterated-below`. We covered these functions in class, but if you need a refresher on how iterators work, look it up in the Help Desk by right-clicking on `cs111/iterated` in the `require` statement at the top of the file, or scroll to the bottom of this assignment and read Appendix 1.

For this part of the homework, we've provided placeholder stubs for your code. Just look for the line that says something like:

```
(define question-1 "fill this in")
```

and replace the string `"fill this in"` with your answer. This will allow the tests to find your code.

Question 1: A Simple Bullseye



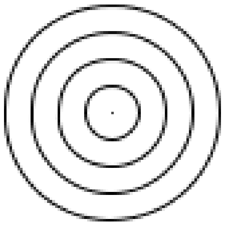
Part (a)

Use `overlay` to generate a bullseye consisting of five concentric circles. The smallest circle should have a radius of 10, and the largest should have a radius of 50.

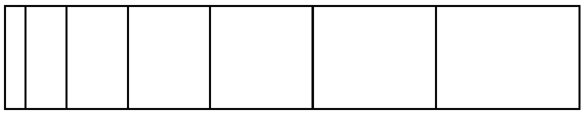
Part (b)

Now use `iterated-overlay` to generate an image identical to your result from Part (a).

Remember that iterators begin counting at 0, not 1, which means you may need to adjust your math slightly to avoid creating a circle with zero radius. If you're getting something like Figure 2, that's probably what's happening to you:

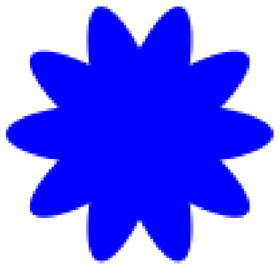


Question 2: A Row of Rectangles



Use `iterated-beside` to generate a row of seven rectangles matching the image above. The heights of all the rectangles should be 50. The widths should range from 10 to 70.

Question 3: A Simple Flower



Use `iterated-overlay` to generate a flower matching the image above. You will need to use five ellipses defined as follows:

```
(ellipse 100 25 "solid" "blue")
```

> *Hint:* look up the help desk documentation for the `rotate` function. Remember that there are 360 degrees in a circle, and you're dividing these by five ellipses.

Interlude: Representing colors with RGB(A)

Thus far, we've only worked with colors in string form, such as `"red"` or `"blue"`.

Colors can also be represented in a format called RGB. The details of RGB are beyond the scope of this course (and possibly also the author's knowledge), but you can think of a color as a list of three values between 0 and 255, corresponding to how red, green, and blue it is.

Here are some example RGB values:

- `"blue"` is `(0 0 255)`. Notice that this color has zero redness, zero greenness, and full blueness.
- `"black"` is `(0 0 0)`. This color has zero redness, greenness, and blueness.
- `"white"` is `(255, 255, 255)`. This color has full redness, greenness, and blueness, which isn't intuitive but just go with it.

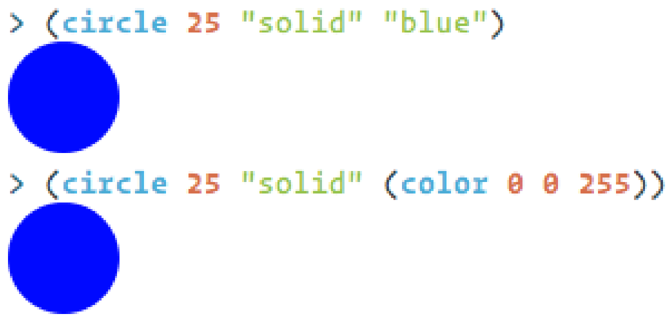
In Racket, we denote RGB colors with the `color` function:

```
;; color : number number number -> color
(color <Redness> <Greenness> <Blueness>)
```

where each of `<Redness>`, etc. is a number between 0 and 255. So we can replace `"blue"` with

```
(color 0 0 255)
```

as demonstrated by the following REPL session:

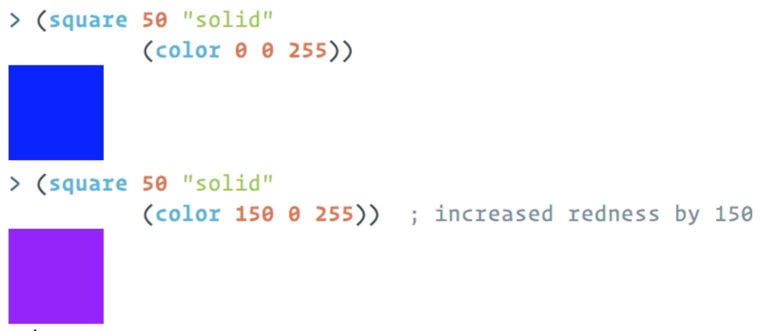


```
> (circle 25 "solid" "blue")

> (circle 25 "solid" (color 0 0 255))
```

Returning to the flower you wrote in the previous question, **change the color** `"blue"` in the ellipse definition to `(color 0 0 255)`. The test should still pass.

Since we can represent colors numerically, we can also use simple math to *change* colors. For example, increasing a color's "R" value by 150 will make it 150 units more red:



```
> (square 50 "solid"
          (color 0 0 255))

> (square 50 "solid"
          (color 150 0 255))  ; increased redness by 150
```

We can also control the *opacity* of a color by adding a fourth number, known as the **alpha value**. The alpha value is also a number between 0 (completely transparent) and 255 (completely opaque). Once you add an alpha value to an RGB color, you're using RGBA format (the A stands for "alpha").

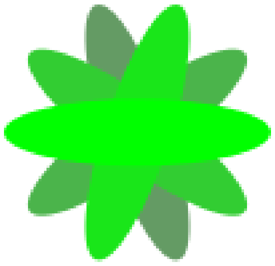
```
> (square 50 "solid"
          (color 0 0 255 0))
```

```
> (square 50 "solid"
          (color 0 0 255 100))
```

```
> (square 50 "solid"
          (color 0 0 255 200))
```

Note that the Racket `color` function takes either three or four numbers. If you only supply three, Racket assumes you want a fully opaque color, so the alpha value will default to 255.

Question 4: A Colorful Flower



Use `iterated-overlay` to generate a flower matching the above image. This flower should be identical to the previous one, except the ellipses should be colored accordingly:

- The first ellipse produced by `iterated-overlay` should be perfectly **green**, i.e. `(color 0 255 0)`.
- Each successive ellipse should be 25 units more red and 25 units less green than the previous ellipse.

  Hint: You must use `iterated-overlay` for this question. You can start by copying your solution from Question 3, then modifying the part of the code responsible for the color of the ellipse on each iteration.

Interlude: `interpolate-colors`

```
> (square 50 "solid"
          (interpolate-colors (color 0 0 255)
                              (color 255 0 0)
                              0.5))
```

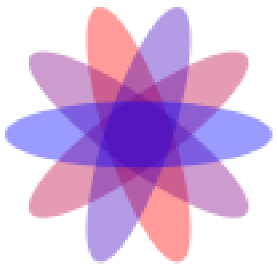
RGB math is hard, let's go shopping! Actually let's not, but manually calculating RGB shade differences can still be irksome. Since the disciplinary purpose of computer science is to help programmers self-justify their laziness, we now introduce a new function called `interpolate-colors`, which provides a much easier way to blend two colors:

```
;; interpolate-colors : color, color, number -> color
(interpolate-colors <Color1> <Color2> <Fraction>)
```

where the `<Fraction>` is a number between 0 and 1, which denotes how much to blend the two colors. Using a fraction of 0 just returns `<Color1>`, and a fraction of 1 just returns `<Color2>`.

ust as `iterated-overlay` abstracts over the tedium of calling `overlay` with ten nearly-identical `circles`, `interpolate-colors` abstracts over the math of computing the RGB difference between two colors.

Question 5: A Fancy Flower



Use `iterated-overlay` and `interpolate-colors` to generate a flower matching the above image. This flower should be similar to the previous one, except that each ellipse should have an alpha value of 100.

You must use `interpolate-colors` in your implementation, starting with blue and ending with red. **Remember that iteration starts at 0, so if you call an iterator n times, the final iteration will be $n - 1$.** You may need to adjust your math to make sure the fifth iteration is completely red.

Since debugging colors is difficult with reduced opacity, we have provided a completely opaque test image for you to use. Search for the line

```
;; (define q5-colors ...)
```

and uncomment. Now you can write your own test to compare your output to this image. Once your colors are interpolating correctly, delete or comment out any extra tests you wrote, and update your answer to incorporate opacity. Make sure your final answer passes the main `question-5` test (it should be identical to the above image).

Part 2: Function Abstractions

Now that you're familiar with the abstractions provided by the `iterated` library, we can start abstracting even more. In this section, you will write reusable functions to generate images from some template, but with even more flexibility.

Whenever you write a function, you should start by writing two comments: a **signature** and a **purpose statement**.

```
;; doubler : number -> number
;; takes a number and multiplies it by two
(define doubler
  (lambda (n) (* 2 n)))
```

The first line is the **signature**, which defines the **name** of the function, followed by a colon, the types of the **input arguments** in order, and the type of the **return value**.

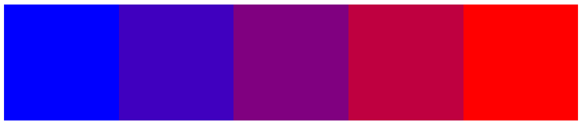
The second line is the **purpose statement**, which is a brief human-readable description of what the function does.

In this assignment, we've provided you with signatures and purpose statements for the first few functions you need to define. Whenever signatures are not provided, you are required to write your own.

In order to let the tests pass, we have commented out the function template and tests for each of these questions. To uncomment out a region, select it with your cursor (you can highlight the whole box as a single item) and in the menu, go to `Racket > Uncomment` to remove the comments.

Question 6: Paint Chips

```
;; swatch : color, color, number -> image
(define swatch
  (lambda (color1 color2 num-squares) ...))
```



Write a function called `swatch` which takes three arguments:

1. A start color,
2. An end color, and
3. A number of squares to generate.

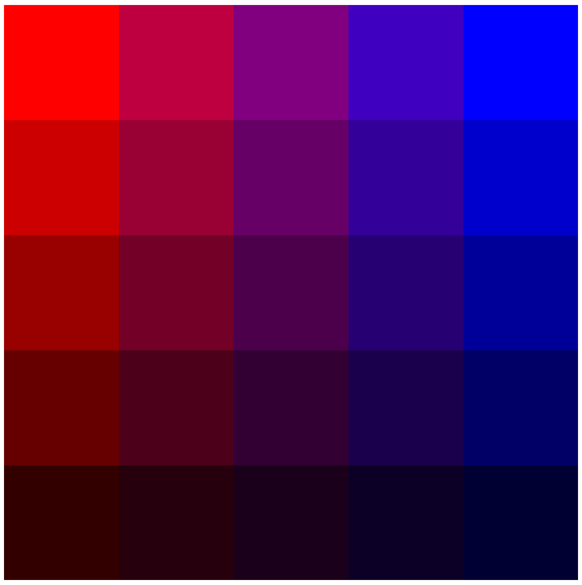
`swatch` should return a single image, consisting of a row of 50x50 squares. The number of squares to generate is given by `num-squares`, the third function argument.

The leftmost square should have color `color1`, and the rightmost square should have color `color2`. The squares in between should evenly interpolate between the two colors. (Again, remember that iteration starts at 0 and ends at $n - 1$.)

For example, calling `(swatch (color 0 0 0) (color 255 255 255) 2)` should produce a row consisting of two squares, one completely black and one completely white.

Question 7: Swatch Grids

```
;; swatch-grid : color, color, number, number -> image
(define swatch-grid
  (lambda (color1 color2 num-rows num-cols)...))
```



Using the `swatch` function you wrote in Question 6, write a function called `swatch-grid` which takes four arguments:

1. A start color,
2. An end color,
3. A row count, and
4. A column count.

`swatch-grid` should return an image, a grid with `num-rows` rows and `num-cols` columns. Each grid cell should be a 50x50 square.

- The upper left square should have color `color1`, and the upper right square should have the color `color2`.
- Each *row* should interpolate evenly between the leftmost and rightmost color, like before.
- Each *column* should interpolate between the topmost color, and black: `(color 0 0 0)`. However, the last column should not be completely black – this is different from what we've previously done. More specifically, if there are m rows, then the last row should stop $1/m$ short of being completely black.

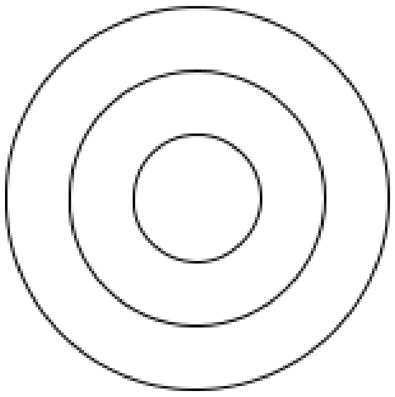
You must use the `swatch` function you wrote in Question 6.

> *Hint:* experiment with the function `iterated-above`. Its signature is identical to `iterated-beside`.

Question 8: Bullseye Revisited

> *Recall that in Question 1, you wrote two equivalent expressions to generate a bullseye consisting of 5 rings with an outer radius of 50. Wouldn't it be great if we could generalize this template to create bullseyes with any radius size and number of rings? (Say yes, you are a computer scientist, abstraction is beautiful, etc.) Well, the great news is that's exactly what you're going to do next!*

```
;; bullseye/simple : number, number, color -> image
(define bullseye/simple
  (lambda (num-rings radius line-color) ...))
```



Write a function called `bullseye/simple` which takes two arguments:

1. A number of rings,
2. A radius, and
3. A color.

`bullseye/simple` should return an image of a bullseye with the specified number of rings. The outermost circle should have the specified radius. The lines of the bullseye should be the given color.

In contrast to previous functions, you must:

1. Write your own **signature** and definition for `bullseye/simple`, and
2. Convert the two images provided in the `.rkt` file into test cases.

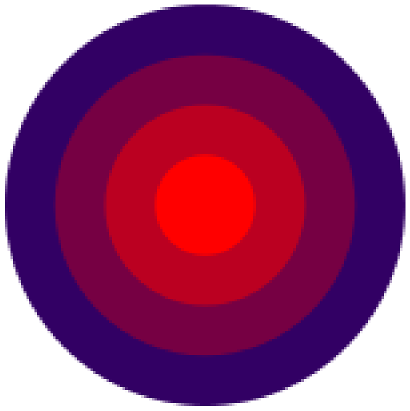
We've provided one complete test case to start you off. Specifically, calling

```
(bullseye/simple 5 50 (color 0 0 0))
```

should produce a bullseye identical to your answer for both parts of Question 1. Compare your code for Questions 1(a), 1(b), and 8, and notice the increasing generalization.

Question 9: Colorful Bullseye

```
;; bullseye/color : number, number, color, color -> image
(define bullseye/color
  (lambda (num-rings radius inner-color outer-color) ...))
```



Write a function called `bullseye/color` which takes four arguments:

1. A number of rings,
2. A radius,
3. An inner color, and
4. An outer color.

`bullseye/color` should return an image of a bullseye with the specified number of rings. The outermost circle should have the specified radius. The innermost ring should have a color given by the third argument, and the outermost ring should have a color given by the fourth argument. The intermediate rings should interpolate evenly between the two colors.

Again, you must write your own function **signature** and definition. You must also convert the two provided images into test cases.

Turning It In

Before turning your assignment in, **run the file one last time** to make sure that it runs properly and doesn't generate any exceptions, and all the tests pass. Assuming they do, press the `EECS 111 Handin` button to submit your assignment.

Appendix 1: A Refresher on Iterators

An **iterator**, broadly speaking, is a program that counts up to a specified number. Each time the count increases is called an **iteration**, and iterations begin with zero (so if you tell an iterator to count for 3 iterations, it will produce the sequence 0, 1, 2). Iterators are used to repeat some action for a predetermined number of times.

Each of the `iterated-` functions in this assignment follow the same general usage pattern. Consider `iterated-overlay`:

```
;; iterated-overlay : (number -> image), number -> image
(iterated-overlay <Generator> <NumberOfIterations>)
```

The signature `(number -> image), number -> image` says that `iterated-overlay` takes two arguments:

1. `(number -> image)`: this means "a function that takes a number and returns an image." Specifically, this is the **generator function** that gets called on each iteration. The number passed to the generator is the current count.
2. `number`: this is the **number of iterations** to complete before the iterator terminates.

When the iterator has counted up the specified number of times, it returns an `image`, which is the accumulated result of the iterations.

So, for instance, if we call

```
(iterated-overlay g 4)
```

is equivalent to

```
(overlay (g 0)
         (g 1)
         (g 2)
         (g 3))
```

In words, `iterated-overlay`:

- Calls `g` four times, passing in the values `0`, `1`, `2`, and `3` in that order, and

- Overlays the four results into a single image (remember, `g` returns an image, so calling `g` four times gives us four images to combine), and
- Returns that combined image.

Appendix 2: Functions vs. Expressions

In Part 1, your answers (e.g. `question-1` and so on) were **expressions** formed by calling other functions. In Part 2, your answers will be **functions** formed using `lambda`.

The following table gives examples of the difference between functions and expressions:

Expression	Function
$2+2$	$f(x)=2+2$
$x+1$	$g(x)=x+1$
$g(3)$	$h(x)=g(x)$
`(circle 50 "solid" "blue")`	`(lambda (a-color) (circle 50 "solid" a-color))`
`(iterated-overlay (lambda (n) (square n "solid" "blue")) 5)`	`(lambda (number-of-iterations) (iterated-overlay (lambda (n) (square n "solid" "blue")) number-of-iterations))`

One way to summarize the difference is that expressions can be simplified to a concrete value of some kind, whereas functions have missing information (the arguments) which need to be filled in before we can simplify. When a function is *called* with that information, we can plug in the missing information and simplify.